On the Naturalness of Proofs

Vincent J. Hellendoorn
University of California, Davis
Davis, California, USA
vhellendoorn@ucdavis.edu

Premkumar T. Devanbu
University of California, Davis
Davis, California, USA
ptdevanbu.edu

Mohammad Amin Alipour
University of Houston
Houston, Texas, USA
alipour@cs.uh.edu

ABSTRACT

Proofs play a key role in reasoning about programs and verification of properties of systems. Mechanized proof assistants help users in developing and checking the consistency of proofs using the proof language developed by the systems; but even then writing proofs is tedious and could benefit from automated insight. In this paper, we analyze proofs in two different proof assistant systems (Coq and HOL Light) to investigate if there is evidence of "naturalness" in these proofs: *viz.*, recurring linguistic patterns that are amenable to language models, in the way that programming languages are known to be. Such models could be used to find errors, rewrite proofs, help suggest dependencies, and perhaps even synthesize (steps of) proofs. We apply state-of-the-art language models to large corpora of proofs to show that this is indeed the case: proofs are remarkably predictable, much like other programming languages. Code completion tools for Coq proofs could save over 60% of typing effort. As proofs have become increasingly central to writing provably correct, large programs (such as the CompCert C compiler), our demonstration that they are amenable to general statistical models unlocks a range of linguistics-inspired tool support.

CCS CONCEPTS

• **Software and its engineering** → *Software verification*; Software maintenance tools; • **General and reference** → *Verification*;

KEYWORDS

Proofs, Deep Learning, Naturalness

ACM Reference Format:
Vincent J. Hellendoorn, Premkumar T. Devanbu, and Mohammad Amin Alipour. 2018. On the Naturalness of Proofs. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3236024.3264832

1 INTRODUCTION

Proofs have played a key role in discovering and capturing knowledge since the dawn of mathematics. The power of formal logic to prove complex theorems in multiple, verifiably correct steps has been available for computer scientists' use, in languages such as Prolog (for first-order logic) and, more recently, *via* interactive frameworks for higher-order logic languages, which can automatically establish many simpler theorems, *e.g.* HOL [1]. Such frameworks have been used to prove tens of thousands of theorems, each of which can in turn be used as dependencies in other theorems. Formal verification of program properties is greatly facilitated by proof languages as well: languages like Coq [3] have been used to prove many program properties, even verifying an entire C compiler (CompCert [11]).

Much like programming, writing proofs is difficult, tedious and time-consuming. Real-world proofs often consist of many steps and may depend on many other theorems. In addition, while most programming language compilers accommodate the incremental development of programs, from partially correct skeletons to fully operational versions, proofs must be universally sound at every step of construction. Developer assistance tools can be of great help given these constraints. At present, these predominantly come in the form of automated sub-theorem proving (for instance, Coq includes keywords like `auto` and `intuition` [3]). Other programming languages have benefited from automated code completion [8, 14], fault localization [13] and rewriting [2]. All of these can be enabled by language models, which capture and utilize typical repetitive patterns in bodies of text. If such patterns also exist in typical proofs, we may bring a substantial range of assistance tools within reach of proof-writing and hopefully simplify this process. In this work, we establish that this is indeed the case by studying two very different corpora of proofs: one written in Gallina, Coq's specification language and one consisting of kernel-level traces of HOL Light proof steps, formatted to resemble typical logical expressions [6, 10]. We show that both Coq and HOL Light proofs are highly predictable, far more so than natural languages. Similar to other programming languages, proofs exhibit high degrees of locality. Our work opens the door for a range of naturalness-supported developer assistance tools for proof-writing.

2 LANGUAGE MODELS

Language models measure the fluency of a sample (test) text by comparing its statistics to those observed in a large (training) corpus of real-world text. For instance, an n-gram language model may collect frequencies of sequences of n adjacent words from a collection of newspaper articles and compare these to sequences that occur in a new article. If the new article is relatively unpredictable given the past data, this indicates that its writing-style is atypical, possibly even containing errors. In addition, a language model may be queried for suggestions in a context; if it adequately "understands" the context, it should recommend useful completions of e.g. a word or sentence.

In practice, n-gram models are often used as baseline models because they are fast to estimate and remarkably accurate. They

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3264832

accomplish this by only considering contexts of $n - 1$ tokens, where n is typically around 5, which tends to be the most informative bit of context. This does ignore many long-range dependencies, however, so deep learning models, based on recurrent neural networks, have more recently come to dominate the state-of-the-art in many natural language modeling tasks. These models abstract context into opaque latent states that theoretically allow them to carry dependencies over long distances (though practically they tend to be more limited). In source code, both types of models have proven highly successful: in general, tokens (*e.g.* punctuation, keywords, identifiers) in programming languages are more predictable than those in natural languages. n-gram models can especially benefit from locality (e.g. tokens in the same file), allowing them to outperform deep learning models in some settings [7].

We use both (dynamic) n-gram and recurrent neural network models to study repetitiveness in proofs. Specifically, we use SLP-Core[1] to implement n-gram models, both plain and with file-cache components. This model uses a scoped n-gram regime, weighting n-gram frequencies from different program scopes.

We implement recurrent neural network language models in CNTK,[2] using a fairly typical architecture with 300-dimensional embeddings and two 650-dimensional, GRU activated hidden layers, with a residual connection. We apply drop-out regularization (50% likelihood of disabling a hidden neuron at training time) and batch-normalization to the second hidden layer. The network is trained across 10 epochs with an initial learning rate of 0.002 per token that is decayed by half every epoch starting epoch 5. On account of the small vocabularies, we found performance to be better when the model was allowed to back-propagate gradients across more time-steps: typically, the recurrence is only "unfolded" for 20 to 50 words, but we achieved substantially better results at 100 steps. Because the corpora are fairly large (HOL's is much larger than typical for code) but the vocabularies are small, we also used relatively large minibatch sizes – 10,000 tokens for Coq and 20,000 for HOL.

We fix the vocabulary for the deep learning model by treating all tokens that are seen only once at training time as a generic "unknown" token. This is necessary because typical RNNs are not able to learn new tokens at test time; instead they must treat these as a generic "unknown" token that must also be learned at training time. We note that the vocabulary is *unlimited* for the n-gram models. We account for this discrepancy by following the approach described by Hellendoorn & Devanbu [7], as described below.

2.1 Metrics

Cross-entropy (typically just called entropy) is our primary metric because it is commonly reported in language modeling work. This information-theoretic metric captures the predictability of a text in terms of the number of additional bits needed to communicate each token (on average) to a receiver if they had access to the same language model. It is computed by averaging the negative log-likelihood across all the tokens in the corpus, which reflects their typical predictability.[3] Lower values imply higher predictability, with the optimal entropy score being 0, at which point a model experiences zero "surprisal" at observing any next token. Typical values for natural languages are in the range of 5 to 8 bits (where 5 bits is currently the state-of-the-art, for neural network models estimated over extremely large corpora) and values for source code are in the range of 1 to 4 bits [7, 8]. Entropy serves well as a general indication of predictability but may not perfectly reflect predictability between languages. This is because it is averaged per token: more verbose languages (like Java) will tend to use more (predictable) syntax-related tokens and may achieve lower per-token entropies than less verbose languages (like Haskell).

```
Theorem sqrt2_not_rational :
 forall p q : nat, q <> 0 -> p * p = 2 * (q * q) -> False.
Proof.
intros p q; generalize p; clear p;
elim q using (well_founded_ind lt_wf).
clear q; intros q Hrec p Hneq;
generalize (neq_O_lt _ (sym_not_equal Hneq));
 intros Hlt_O_q Heq.
apply (Hrec (3 * q - 2 * p)
(comparison4 _ _ Hlt_O_q Heq) (3 * p - 4 * q)).
apply sym_not_equal; apply lt_neq;
apply plus_lt_reg_l with (2 * p);
 rewrite <- plus_n_O; rewrite <- le_plus_minus; auto with *.
apply new_equality; auto.
Qed.
```

Figure 1: An example of a proof in Coq, proving the irrationality of the square root of 2.

```
...
(!a. (!b. (!m. (!n. ((n <= m) ==>
(((real_add ((float a) (int_neg (int_of_num m)))) ((float b)
(int_neg (int_of_num n)))) = ((float ((int_add ((int_mul
...
```

Figure 2: A proof snippet in HOL Light

Prediction accuracy allows us to approximate how useful a code completion tool could be for a language by recommending a list of tokens in every context. Typically, accuracy is reported based on where the correct item appears in an ordered list of suggestions. Top-1 accuracy captures how often the correct item is also the top-ranked item (we will mainly report this metric); top-5 and top-10 accuracy metrics allow the suggestion to appear anywhere in that respective range to be counted correct and thus implicitly assume that a user will look at those top-k items before choosing a completion or typing the token themselves. Often, the most simple, shortest tokens are also the most predictable, so that top-k accuracy scores may be deceptive; it is unlikely that a programmer will ask for a 1-character completion. We will also report the relative *character savings* at top-1/5/10: the percentage of characters in the file that a programmer would save (not have to type) when the correct item is in the top-k suggestions.

3 CORPORA

To capture the types of proofs that are commonly written, we use two major data sources consisting of Coq proofs, and higher-order logic proofs. Table 1 shows the characteristics of our datasets.

Coq is a proof assistant for logic based on Calculus of Inductive Constructions [4]. It uses the Gallina language for specification of theorems and proofs. A proof in Coq can be written as a sequential application of hypotheses or lemmas, enclosed between `Proof` and `Qed` commands, that will prove the proof goal. Coq allows defining

[1] https://github.com/SLP-team/SLP-Core

[2] https://cntk.ai/

[3] Specifically, the geometric mean probability, which is more stable than the arithmetic mean given the wide range of probabilities that may be encountered.

Table 1: Statistics of the datasets used in our work

Dataset	# files	# tokens	# vocabulary
Coq	3,589	9,817,678	37,813
HOL	11,410	107,576,235	1,837

custom tactics to bundle common proof steps together for reuse, as well as sophisticated proof search based on available lemmas and proof subgoals. Figure 1 shows a proof for irrationality of $\sqrt{2}$ in Coq taken from the Coq manual [3]. Dots (`.`) and semicolons (`;`) demarcate the steps in the proofs. This Gallina script uses proof tactics such as `intro` and `rewrite`, and applies previous theorems/lemmas, such as `sym_not_equal` in the proof. We used 17 active (based on the number of proof scripts and commits) Coq projects on GitHub for this study. The Gallina files in these projects were randomly split into a 90% train and 10% test split for our language models.

Higher Order Logic (HOL) allows universal ($\forall$) and existential ($\exists$) quantifiers over higher-order objects; where first-order objects describe individual objects in the domain, second-order objects describe sets of objects, third-order objects describe sets of sets of objects, and so on. HOL can thus describe highly complex relations among objects in domains, making it very expressive. Figure 2 shows a proof snippet in HOL.

Kaliszyk *et al.,* collected the HOLStep dataset, consisting of processed trace data of real proofs from the HOL Light [6] kernel. This produced a dataset of proofs, each starting with some (typically simple) dependencies followed by a number of "positive" (useful) and "negative" (not useful) steps. The original proof contained the positive steps only, and the task of a machine learner in this challenge was to decide which steps were real and which were fake. The authors registered some reasonable success rates (ca. 80%) using convolutional neural networks, suggesting that there is indeed something "natural" about real proof steps.[4] We are not interested in the challenge task itself, but instead use the positive steps from their proofs as training data. This gives us 10 thousand training proofs (with nearly 99 million tokens) and 1,410 test proofs (with ca. 8.5 million tokens).

The steps in the HolStep dataset are based on kernel-level traces (for simplicity, we will still refer to them as HOL Light proofs) and were all tokenized to resemble “textbook-style mathematics”, making these proofs very different from the Coq dataset, which makes our findings more broadly appicable. Since these are not human-written proof steps, code completion is unlikely to be useful here, but entropy and general predictability are. Note also that HolStep contains proofs that use only primitive higher order inferences. That is, in contrast to proofs in Coq, it does not contain any high-level proof tactics. We reiterate that we do not compare the predictability of these languages (in part because of these differences); we instead show that both are remarkably natural.

4 RESULTS

We present the entropy results for our various models in Table 2. We also compare the predictability (and aspects of it) of both proof languages to other programming languages.

Table 2: Entropies (in bits) and top-1 prediction accuracy on the proof corpora. Character savings gives an indication of typing effort reduction, using the top completion if correct.

	Entropy		**Top-1 Acc.**		**Char. savings**	
	Coq	HOL	Coq	HOL	Coq	HOL**
base *n*-gram	3.52	2.49	55.7%	55.5%	46.9%	56.1%
n-gram+cache	2.28	1.28	70.2%	78.8%	63.0%	80.8%
RNN	3.03	1.72	56.6%	68.6%	46.5%	67.9%
RNN open*	–	–	56.2%	68.6%	45.6%	67.9%

*RNN with open vocabulary at test time, for which suggesting the generic unknown token is not considered accurate

**HOLStep data is based on kernel traces, not human written steps; typing effort is thus only shown for reference.

4.1 Performance Characteristics

Looking first at the per-token entropy values for simple *n*-gram models, both Coq and HOL Light (kernel-level) proofs are relatively predictable compared to natural languages, whose *n*-gram entropies tend to be around 6 to 8 bits per token. Coq's plain *n*-gram entropy of 3.52 bits/token is quite similar to typical entropies for Java [8]. HOL Light's entropy is substantially lower, although we caution the reader against assigning too great an importance to the difference for several reasons: (1) HOL Light's vocabulary is much smaller (and its training corpus larger), (2) the entropy is measured per-token, so that more verbose languages tend to score lower (e.g. Java is less entropic than Python by this measure), and (3) we do not compare these two languages; we just aim to understand their characteristics in the larger context of programming language "naturalness".

The RNN model outperforms the plain *n*-gram model by a sizable margin, similar to findings in prior work [17]. It is especially good at modeling HOL Light proofs, which may be due to their very small vocabulary; this language may be particularly amenable to RNNs. RNN prediction accuracy is also better in HOL Light but is about the same as the plain *n*-gram models for Coq. This matches insights from prior work, which found that the RNN mainly achieves low average entropy by being very confident in correct predictions (thus achieving effectively zero entropy on those), whereas the *n*-gram model is more "reserved" due to smoothing. As a result, the entropy gain of the RNN over the plain *n*-gram model may just be concentrated on the most predictable tokens and not translate into any useful improvements in prediction accuracy. The RNN model is in turn outperformed by an *n*-gram model with a cache component, as also found in previous work on modeling Java code [7]. This holds for both entropy and prediction accuracy. The RNN does use a limited vocabulary, whereas the *n*-gram model vocabulary is unlimited; we will discuss this more in Section 4.2.

Completion accuracy on Coq shows artifacts of inflation relative to actual character savings: less typing effort is saved than suggested by the top-1 accuracy alone across all models. This means that relatively short tokens are more predictable while offering completions on those is unlikely to save a developer effort. However, the difference is relatively small, possibly because very few long (over 10 characters) tokens were found in the Coq data, whereas other programming languages will often contain many more such tokens. Performance on the HOLStep dataset did not show a discrepancy at all, in fact yielding slightly better character savings than prediction accuracy. However, this dataset did not consist of human-written

[4] Although we found negative examples in this dataset to be different in entropy than positive ones, which may allow deep neural networks to distinguish between positive and negative steps without any insight into the actual proof

steps but rather of trace data, so that the typing effort reduction is less relevant than characterization metrics such as entropy.

4.2 Vocabulary

The vocabulary of both proof systems is remarkably low, much lower than both natural languages and programming languages considering their corpus sizes. For instance, a 16M token Java corpus in prior work had a vocabulary of ca. 200,000 tokens [7], while the Coq corpus (at 10M tokens) has less than 38,000. The HOL Light proofs' even smaller vocabulary is mainly due to variable names being lost in the trace data.

Relatively small vocabularies tend to imply higher predictability, but we do note that this *per se* by no means explains the low entropy values. Even a vocabulary of a thousand tokens would yield an entropy of ca. 10 bits if those tokens were used at random. Furthermore, the popular PTB English dataset has a vocabulary of just 10,000 words and its best entropy values are around 6 bits. Instead, the vocabulary size may justify some of the lower entropy values of HOL compared to Coq, but a substantial inherent repetitiveness is responsible for the overall low entropy values.

The deep neural network uses a closed vocabulary when training and testing, meaning some tokens are treated as generic "unknowns". As a result, it uses a vocabulary of 24,427 tokens for the Coq data and 1,759 for the HOL data. This necessarily inflates its performance somewhat as these unknown tokens are fairly common and easy to predict while replacing tokens that are rare and hard to predict. Following Hellendoorn & Devanbu, we include an additional row in Table 2 that instead gives no reward for predicting the unknown token [7]. As can be seen, this effect is quite mild in our data; prediction accuracy on Coq decreases somewhat (especially in terms of character savings because rare tokens tend to be longer), but not as much as in prediction of Java in prior work. Performance on HOL Light proofs does not change significantly at all because of its naturally almost completely closed vocabulary. Thus, proof languages may be a domain where the deep neural network's difficulty learning new tokens is not such a great disadvantage.

4.3 Locality

Both Coq and HOL n-gram entropies benefit substantially from a cache component: it more than halves surprisal in both,[5] yielding an especially impressive improvement in HOL considering the already low entropy. Programming languages have been shown to exhibit a high degree of locality in their patterns, which is especially present in locally typical use of identifiers (e.g. only within one file, package or project) [7, 15]. However, HOL's vocabulary is already extremely small, implying that there is also a high degree of locality in the organization of the arguments in the proofs themselves. To ensure that this is not just a matter of vocabulary repetition, we also compared with a 1-gram cache (a cache that only stores tokens, not sequences): this model achieves an entropy of 2.41 bits/token, which is substantially worse than what can be achieved by caching whole sequences. Both datasets see a substantial increase in prediction accuracy and (even more so) relative character savings of the top-1 predictions. Since the longest tokens in most proofs are references to dependencies (other theorems), the cache component appears to successfully pick up on repeated use of those dependencies, which should be helpful for a developer in a code completion setting.

5 IMPLICATIONS

Proof assistants have traditionally mainly been used by theory-savvy users. With the increasing demand for certified software, there are increasingly more developers who need to use proof assistants such as Coq or HOL to prove properties of their systems and generate certified code. Naturalness of proofs may provide new opportunities to support them.

Guiding proofs: Naturalness of source code has been key in devising useful tools for programmers, e.g. code completion [14], based purely on the textual representation of the source code. Our results suggest that proof scripts are also natural, making it possible to provide proof guidance by predicting next tokens and perhaps entire statements in proof scripts based on surrounding context.

Unnatural approaches: Proving theorems can be a tedious job. Usually, one tries many different deconstructions of a proof goal into subgoals and multiple steps to prove them before finding the right approach. One way to address this issue may be to alert the user whenever the proof script starts to deviate from natural proofs. Studies on source code have shown that buggy code is unnatural [13], and similarly "surprising" proof steps may help developers catch bugs early.

6 RELATED WORK

Naturalness of Code Hindle *et al.,* showed that the degree of repetitive patterns in source code reflects and exceeds repetitiveness in natural language [8]. This naturalness facilitates applications such as code completion [14], error checking, variable name recovery in minified Javascript [16] and style checking [2]. Proof scripts are themselves programs that are interpreted by proof assistants, making them much like classical programming languages. We studied two large corpora of proof scripts and observed that they are indeed also natural.

Guiding Proof Search The process of proving a conjecture is essentially a search in the space of possible sequences of application of lemmas and hypotheses. This search space can be prohibitively large; thus, proof assistants attempt to guide this search by hinting theorems that are more likely to lead to the proof. Premise selection techniques attempt to find the most relevant theorems and lemma to proof goals and are traditionally driven by hand-made heuristics [9]. More recently, with advent of large corpora of proofs, machine learning techniques have been employed to find related theorems given a conjecture, though this has been largely limited to choosing between a useful and unused next step [5, 10, 12]. Language models, on the other hand, can be used for a wide range of tasks, both generative (e.g. synthesis, translation, completion) and descriptive (e.g. fault localization, convention checking) in nature.

ACKNOWLEDGEMENTS

This work was supported by the National Science Foundation, award number 1414172

REFERENCES

[1] HOL Interactive Theorem Prover. https://hol-theorem-prover.org/#about.

[5] every bit of entropy is a factor two increase in geometric mean probability

[2] Allamanis, M., Barr, E. T., Bird, C., and Sutton, C. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014), ACM, pp. 281–293.

[3] Coquand, T., and Huet, G. The Coq Proof Assistant. https://coq.inria.fr/.

[4] Coquand, T., and Huet, G. The calculus of constructions. *Information and computation 76*, 2-3 (1988), 95–120.

[5] Graham-Lengrand, S., and Färber, M. Guiding smt solvers with monte carlo tree search and neural networks. *AITP 2018* (2018).

[6] Harrison, J. Hol light: A tutorial introduction. In *International Conference on Formal Methods in Computer-Aided Design* (1996), Springer, pp. 265–269.

[7] Hellendoorn, V. J., and Devanbu, P. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (2017), ACM, pp. 763–773.

[8] Hindle, A., Barr, E. T., Su, Z., Gabel, M., and Devanbu, P. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on* (2012), IEEE, pp. 837–847.

[9] Hoder, K., and Voronkov, A. Sine qua non for large theory reasoning. In *International Conference on Automated Deduction* (2011), Springer, pp. 299–314.

[10] Kaliszyk, C., Chollet, F., and Szgedy, C. Holstep : a Machine Learning Dataset Higher - Order Logic Theorem Proving. *ICLR* (2017), 1–12.

[11] Leroy, X., et al. The compcert verified compiler. *Documentation and userâĂŹs manual. INRIA Paris-Rocquencourt* (2012).

[12] Loos, S., Irving, G., Szegedy, C., and Kaliszyk, C. Deep Network Guided Proof Search. In *21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning Deep* (2017), vol. 46, pp. 85—-105.

[13] Ray, B., Hellendoorn, V., Godhane, S., Tu, Z., Bacchelli, A., and Devanbu, P. On the "naturalness" of buggy code. In *Proceedings of the 38th International Conference on Software Engineering* (New York, NY, USA, 2016), ICSE ’16, ACM, pp. 428–439.

[14] Raychev, V., Vechev, M., and Yahav, E. Code completion with statistical language models. In *Acm Sigplan Notices* (2014), vol. 49, ACM, pp. 419–428.

[15] Tu, Z., Su, Z., and Devanbu, P. On the localness of software. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2014), FSE 2014, ACM, pp. 269–280.

[16] Vasilescu, B., Casalnuovo, C., and Devanbu, P. Recovering clear, natural identifiers from obfuscated js names. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (2017), ACM, pp. 683–693.

[17] White, M., Vendome, C., Linares-Vásquez, M., and Poshyvanyk, D. Toward deep learning software repositories. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on* (2015), IEEE, pp. 334–345.